

# *Práctica 1*

---

## *Descubriendo el Entorno de Trabajo*

---

### **1.1. Objetivos**

Este laboratorio está pensado como un refuerzo de la asignatura de Fundamentos de Computadores. Permitirá al alumno afianzar sus conocimiento sobre del funcionamiento de un computador con arquitectura Von Neumann. Utilizaremos como componente principal un procesador de ARM, concretamente el ARM7TDMI. En esta primera práctica introduciremos los conceptos básicos de la programación en ensamblador y trataremos de familiarizarnos con el entorno de desarrollo y la arquitectura de la cpu utilizada. En concreto, los objetivos que perseguimos en esta práctica son:

- Adquirir práctica en el manejo del repertorio de instrucciones ARM, incluyendo saltos y accesos a memoria.
- Familiarizarse con el entorno de desarrollo y las herramientas GNU para ARM, ensamblador, enlazador y depurador esencialmente.
- Comprender la estructura de un programa en ensamblador y el proceso de generación de un binario a partir de éste.

### **1.2. Introducción al funcionamiento de un computador**

Un computador, como máquina electrónica que es, sólo entiende señales eléctricas, lo que en electrónica digital se corresponde con apagado y encendido. Por lo tanto, el alfabeto capaz de ser comprendido por un computador se corresponde con dos dígitos: el 0 y el 1 (alfabeto binario).

Las órdenes que queramos proporcionar al computador serán un conjunto de 0s y 1s con un significado conocido de antemano, que el computador podrá decodificar para realizar su funcionalidad. El nombre para una orden individual es instrucción. Programar un computador a base de 0s y 1s (lenguaje máquina) es un trabajo muy laborioso y poco gratificante. Por lo que se ha inventado un lenguaje simbólico (lenguaje ensamblador) formado por órdenes sencillas que se pueden traducir de manera directa al lenguaje de 0s y 1s que entiende el computador. El lenguaje ensamblador requiere que el programador escriba una línea para cada instrucción que desee que la máquina ejecute, es un lenguaje que fuerza al programador a pensar como la máquina. Si se puede escribir un programa que traduzca órdenes sencillas

<b>C = A + B;</b>	Lenguaje de alto nivel
<b>COMPILEADOR</b>	
<b>ldr R1, A</b> <b>ldr R2, B</b> <b>add R3, R1, R2</b> <b>str R3, C</b>	Lenguaje ensamblador
<b>ENSAMBLADOR + ENLAZADOR</b>	
<b>e51f1014</b> <b>e51f2014</b> <b>e0813002</b> <b>e50f3018</b>	Lenguaje máquina (hexadecimal)

Tabla 1.1: Proceso de generación de órdenes procesables por un computador

(lenguaje ensamblador) a ceros y unos (lenguaje máquina), ¿qué impide escribir un programa que traduzca de una notación de alto nivel a lenguaje ensamblador? Nada. De hecho, actualmente la mayoría de los programadores escriben sus programas en un lenguaje, que podíamos denominar más natural (lenguaje de alto nivel: C, pascal, FORTRAN...). El lenguaje de alto nivel es más sencillo de aprender e independiente de la arquitectura hardware sobre la que se va a terminar ejecutando. Estas dos razones hacen que desarrollar cualquier algoritmo utilizando la programación de alto nivel sea mucho más rápido que utilizando lenguaje ensamblador. Los programadores de hoy en día deben su productividad a la existencia de un programa que traduce el lenguaje de alto nivel a lenguaje ensamblador, a ese programa se le denomina compilador.

### 1.3. Introducción a la Arquitectura ARM

Los procesadores de la familia ARM tienen una arquitectura RISC de 32 bits, ideal para realizar sistemas empotrados de elevado rendimiento y reducido consumo, como teléfonos móviles, PDAs, consolas de juegos portátiles, etc. Tienen las características principales de cualquier RISC:

- Un banco de registros.
- Arquitectura *load/store*, es decir, las instrucciones aritméticas operan sólo sobre registros, no directamente sobre memoria.
- Modos de direccionamiento simples. Las direcciones de acceso a memoria (*load/store*) se determinan sólo en función del contenido de algún registro y el valor de algún campo de la instrucción (valor inmediato).
- Formato de instrucciones uniforme. Todas las instrucciones ocupan 32 bits con campos del mismo tamaño en instrucciones similares.

La arquitectura ARM sigue un modelo Von Neumann con un mismo espacio de direcciones para instrucciones y datos. Esta memoria es direccionable por byte y está organizada en palabras de 4 bytes.

En modo usuario son visibles 15 registros de propósito general (R0-R14), un registro de contador de programa (PC), que en ciertas circunstancias puede emplearse como si fuese de propósito general (R15), y un registro de estado (CPSR). Todas las instrucciones pueden direccionar y escribir cada uno de los 16 registros en cada momento. El registro de estado, CPSR, debe ser manipulado por medio de instrucciones especiales.

Aunque el registro contador de programa pueda ser empleado en una instrucción como si se tratase de uno de propósito general, es preciso tener en cuenta los efectos laterales que ello pueda ocasionar. Si se escribe sobre él, se producirá un salto en el flujo de instrucciones del programa. Además, debido al diseño interno del procesador cuando se realiza la lectura de este registro, el valor proporcionado es el de la dirección de la instrucción que lee el PC más 8, es decir, dos instrucciones después de la actual.

El registro de estado, CPSR, almacena información adicional necesaria para determinar el estado del programa, por ejemplo el signo del resultado de alguna operación anterior o el modo de ejecución del procesador. Es el único registro que tiene restricciones de acceso. Está estructurado en campos con un significado bien definido: *flags* (*f*), *reservados* (no se usan en ARM v4) y *control* (*c*), como ilustra la Figura 1.1. El campo de *flags* contiene los indicadores de condición y el campo de *control* contiene distintos bits que sirven para controlar el modo de ejecución. La Tabla 1.2 describe el significado de cada uno de los bits de estos campos. Los que bits están reservados para uso futuro no son modificables y siempre se leen como cero. Los indicadores de condición, bits N, Z, C y V, son modificables en modo usuario, mientras que los bits I, F y M sólo son modificables en los modos privilegiados.

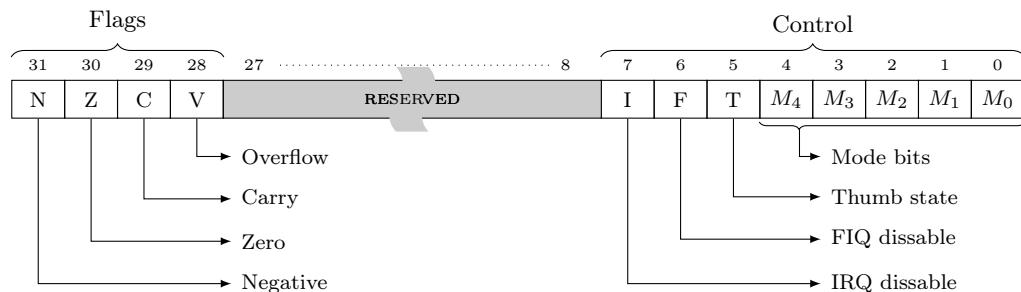


Figura 1.1: Registro de Estado del ARM v4 (CPSR).

Tabla 1.2: Descripción del campo de flags del CPSR.

Bit	Significado
N	Indica si la última operación dio como resultado un valor negativo (N = 1) o positivo (N = 0).
Z	Se activa (Z = 1) si el resultado de la última operación fue cero, de lo contrario permanece inactivo (Z = 0).
C	Su valor depende del tipo de operación: <ul style="list-style-type: none"> <li>Para una suma o una comparación (CMP), C = 1 si hubo <i>carry</i>.</li> <li>Para las operaciones de desplazamiento, toma el valor del bit saliente.</li> </ul>
V	En el caso de una suma o una resta, V = 1 indica que hubo un <i>overflow</i> .

## 1.4. Repertorio de instrucciones

El repertorio de instrucciones de ARM es relativamente grande, y a medida que han ido apareciendo nuevas versiones se han ido añadiendo algunas instrucciones nuevas sobre el repertorio ARM básico. Por motivos pedagógicos, en este laboratorio vamos a limitarnos a estudiar un subconjunto del repertorio básico de ARM, válido en la mayor parte de sus procesadores.

Como es habitual, dividiremos las instrucciones en seis grupos:

- Aritmético-lógicas
- Multiplicación
- Acceso a Memoria
- Salto

### 1.4.1. Instrucciones aritmético-lógicas.

En este apartado veremos las instrucciones que usan la unidad aritmético-lógica (ALU) del procesador. No incluimos aquí las instrucciones de multiplicación, que veremos en el siguiente apartado, debido a que se ejecutan en otro módulo.

La sintaxis de las instrucciones aritmético-lógicas es la siguiente:

`Instrucción{S} Rd, Rn, N      @ Rd ← Rn Oper N`

Donde:

- **Instrucción:** alguno de los mnemotécnicos de la tabla 1.3
- **S:** si se incluye este campo la instrucción modifica los indicadores (*flags*) de condición de CPSR.
- **Rd:** registro destino (donde se almacena el resultado)
- **Rn:** registro fuente (primer operando). Todas las instrucciones menos MOV.
- **N:** segundo operando, denominado *shifter\_operand*. Es muy versátil, pero de momento nos conformaremos con dos posibles modos de direccionamiento: un registro o un inmediato.

Como podemos ver, además de escribir el resultado en el registro destino, cualquier instrucción puede modificar los flags de CPSR si se le añade como sufijo una S.

La Tabla 1.3 resume las instrucciones aritmético-lógicas más comunes. La mayor parte son instrucciones de dos operandos en registro que escriben su resultado en un tercer registro. Algunas como MOV tienen sólo un operando. Otras, como CMP, no escriben el resultado en registro, sólo modifican los bits del CPSR (en estos casos no es necesario indicarlo mediante S).

Tabla 1.3: Instrucciones aritmético-lógicas comunes. ShiftOp representa el *shifter\_operand*.

Mnemo	Operación	Acción
AND	AND Lógica	$Rd \leftarrow Rn \text{ AND ShiftOp}$
ORR	OR Lógica	$Rd \leftarrow Rn \text{ OR ShiftOp}$
EOR	OR exclusiva	$Rd \leftarrow Rn \text{ EOR ShiftOp}$
ADD	Suma	$Rd \leftarrow Rn + \text{ShiftOp}$
SUB	Resta	$Rd \leftarrow Rn - \text{ShiftOp}$
RSB	Resta inversa	$Rd \leftarrow \text{ShiftOp} - Rn$
ADC	Suma con acarreo	$Rd \leftarrow Rn + \text{ShiftOp} + \text{Carry Flag}$
SBC	Resta con acarreo	$Rd \leftarrow Rn - \text{ShiftOp} - \text{NOT(Carry Flag)}$
RSC	Resta inversa con acarreo	$Rd \leftarrow \text{ShiftOp} - Rn - \text{NOT(Carry Flag)}$
CMP	Comparar	Hace $Rn - \text{ShiftOp}$ y actualiza los flags de CPSR convenientemente
CMN	Comparar negado	Hace $Rn + \text{ShiftOp}$ y actualiza los flags de CPSR convenientemente
MOV	Mover entre registros	$Rd \leftarrow \text{ShiftOp}$
MVN	Mover negado	$Rd \leftarrow \text{NOT ShiftOp}$
BIC	Borrado de bit ( <i>Bit Clear</i> )	$Rd \leftarrow Rn \text{ AND NOT(ShiftOp)}$

## Ejemplos

```

ADD  R0, R1, #1      @ RO = R1 + 1
ADD  R0, R1, R2      @ RO = R1 + R2
BIC  R4, #0x05      @ Borra los bits 0 y 2 de R4
MOV   R11,#0         @ Escribe cero en R11
SUB   R3, R2, R1      @ R3 = R2 - R1
SUBS  R3, R2, R1      @ R3 = R2 - R1. Modifica los flags del registro de
                      @ estado en función del resultado
ADDEQ R7, R1, R2      @ Si el bit Z de CPSR está activo R7 = R1 + R2

```

### 1.4.2. Instrucciones de multiplicación

Hay diversas variantes de la instrucción de multiplicación debido a que al multiplicar dos datos de  $n$  bits necesitamos  $2n$  bits para representar el resultado, y a que se dispone de multiplicaciones con y sin signo. Las principales variantes se describen en la Tabla 1.4. Si se les pone el sufijo *S* modificarán además los bits *Z* y *N* del registro de estado de acuerdo con el valor y el signo del resultado. Los operandos siempre están en registros y el resultado se almacena también en uno o dos registros, en función de su tamaño (32 o 64 bits).

## Ejemplos

```

MUL   R4, R2, R1      @ R4 = R2 x R1
MULS  R4, R2, R1      @ R4 = R2 x R1, modifica los flags del registro
                      @ de estado en función del resultado
MLA   R7, R8, R9, R3    @ R7 = R8 x R9 + R3
SMULL R4, R8, R2, R3    @ R4 = [R2 x R3]31..0
                      @ R8 = [R2 x R3]63..32
UMULL R6, R8, R0, R1    @ R8/R6 = R0 x R1

```

Tabla 1.4: Instrucciones de multiplicación.

Mnemotécnico	Operación
<b>MUL</b> Rd, Rm, Rs	$Rd \leftarrow (Rm * Rs) [31..0]$ .
<b>MLA</b> Rd, Rm, Rs, Rn	$Rd \leftarrow (Rn + Rm * Rs) [31..0]$ .
<b>SMULL</b> RdLo, RdHi, Rm, Rs	$RdHi \leftarrow (Rm * Rs) [63..32]$ y $RdLo \leftarrow (Rm * Rs) [31..0]$ , donde los operandos son de 32 bits con signo.
<b>UMULL</b> RdLo, RdHi, Rm, Rs	$RdHi \leftarrow (Rm * Rs) [63..32]$ y $RdLo \leftarrow (Rm * Rs) [31..0]$ , donde los operandos son de 32 bits sin signo.
<b>SMLAL</b> RdLo, RdHi, Rm, Rs	$RdHi \leftarrow (RdHi : RdLo + Rm * Rs) [63..32]$ y $RdLo \leftarrow (RdHi : RdLo + Rm * Rs) [31..0]$ , donde los operandos son de 32 bits con signo.
<b>UMLAL</b> RdLo, RdHi, Rm, Rs	$RdHi \leftarrow (RdHi : RdLo + Rm * Rs) [63..32]$ y $RdLo \leftarrow (RdHi : RdLo + Rm * Rs) [31..0]$ , donde los operandos son de 32 bits sin signo.

**UMLAL** R5, R8, R0, R1       $\text{@ R8/R5} = R0 \times R1 + R8/R5$

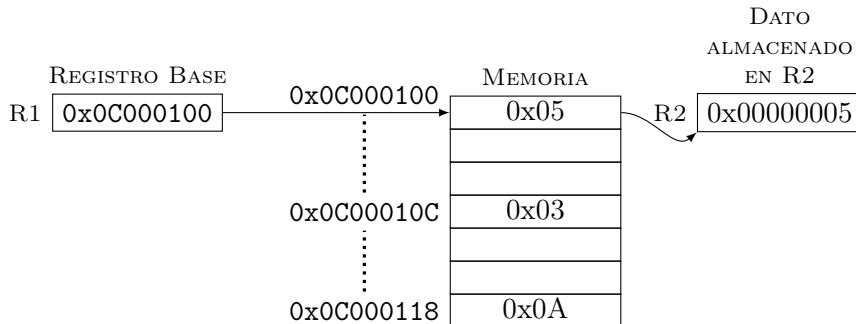
#### 1.4.3. Instrucciones de acceso a memoria (Load y Store)

Las instrucciones de **load** (LDR) se utilizan para cargar un dato de memoria sobre un registro. Las instrucciones de **store** (STR) realizan la operación contraria, copian en memoria el contenido de un registro. En ARMv4 existen varios modos de direccionamiento para las instrucciones **ldr/str**, sin embargo, en esta práctica nos centraremos sólo en los dos más utilizados:

- **Indirecto de registro** La dirección de memoria a la que deseamos acceder se encuentra en un registro del banco de registros. El formato en ensamblador sería:

```
LDR Rd, [Rb] @ Rd ← Memoria( Rb )
STR Rf, [Rb] @ Rf → Memoria( Rb )
```

En la Figura 1.2 se ilustra el resultado de realizar la operación **ldr r2, [r1]**.

Figura 1.2: Ejemplo de ejecución de la instrucción **ldr r2, [r1]**.

- Indirecto de registro con desplazamiento inmediato** La dirección de memoria a la que deseamos acceder se calcula sumando una constante a la dirección almacenada en un registro del banco de registros, que llamamos registro base. El desplazamiento se codifica como valor inmediato (en la propia instrucción) con 12 bits y en convenio de complemento a 2. El formato en ensamblador sería:

```
LDR Rd, [Rb, #desp] @ Rd ← Memoria( Rb + desp )
STR Rf, [Rb, #desp] @ Rf → Memoria( Rb + desp )
```

En la Figura 1.3 se ilustra el resultado de realizar la operación `ldr r2,[r1,#12]`.

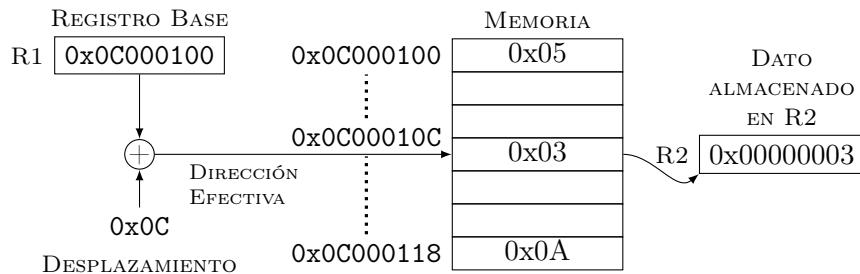


Figura 1.3: Ejemplo de ejecución de la instrucción `ldr r2,[r1,#12]`.

## Ejemplos

```
LDR R1, [R0]           @ Carga en R1 lo que hay en Memoria(R0)
LDR R8, [R3, #4]        @ Carga en R8 lo que hay en Memoria(R3 + 4)
LDR R12, [R13, #-4]      @ Carga en R12 lo que hay en Memoria(R13 - 4)
STR R2, [R1, #0x100]     @ Almacena en Memoria(R1 + 256) lo que hay en R2
```

### 1.4.4. Instrucciones de Salto

La instrucción explícita de salto es Branch (B). La distancia a saltar está limitada por los 24 bits con los que se puede codificar el desplazamiento dentro de la instrucción (operando inmediato). La sintaxis es la siguiente (los campos que aparecen entre llaves son opcionales):

`B{Condición} Desplazamiento @ PC ← PC + Desplazamiento`

donde Condición indica una de las condiciones de la tabla 1.5 (EQ,NE,GE,GT,LE,LT,...) y Desplazamiento es un valor inmediato con signo (precedido de #) que representa un desplazamiento respecto del PC.

La dirección a la que se salta debe ser codificada como un desplazamiento relativo al PC. Sin embargo, como veremos a la hora de programar en ensamblador utilizaremos etiquetas y serán el ensamblador y el enlazador los encargados de calcular el valor exacto del desplazamiento.

Los saltos condicionales se ejecutan solamente si se cumple la condición del salto. Una instrucción anterior tiene que haber activado los indicadores de condición del registro de estado. Normalmente esa instrucción anterior es CMP (ver instrucciones aritméticas), pero puede ser cualquier instrucción con sufijo S.

En el caso de que no se cumpla la condición, el flujo natural del programa se mantiene, es decir, se ejecuta la instrucción siguiente a la del salto.

Finalmente, debemos hacer notar que se puede provocar también un salto escribiendo una dirección absoluta en el contador de programa, por ejemplo con la instrucción **MOV**.

Tabla 1.5: Condiciones asociadas a las instrucciones.

Mnemotécnico	Descripción	Flags
EQ	Igual	Z=1
NE	Distinto	Z=0
HI	Mayor que (sin signo)	C=1 & Z=0
LS	Menor o igual que (sin signo)	C=0 or Z=1
GE	Mayor o igual que (con signo)	N=V
LT	Menor que con signo	N!=V
GT	Mayor que con signo	Z=0 & N=V
LE	Menor o igual que (con signo)	Z=1 or N!=V
(vacío)	Siempre (incondicional)	

## Ejemplos

```
B    etiqueta    @ Salta incondicional a etiqueta
BEQ  etiqueta  @ Salta a etiqueta si Z=1
BLS  etiqueta  @ Salta a etiqueta si Z=1 o N=1
BHI  etiqueta  @ Salta a etiqueta si C=1 y Z=0
BCC  etiqueta  @ Salta a etiqueta si C=0
MOV  PC, #0    @ R15 = 0, salto absoluto a la dirección 0
```

### 1.4.5. Estructuras de control de flujo en ensamblador

Con las instrucciones presentadas hasta ahora podemos implementar cualquier algoritmo. Resulta conveniente pararse un momento a pensar cómo podemos codificar las estructuras básicas de control de flujo que hemos aprendido en los cursos de programación estructurada. Si lo hacemos nos daremos cuenta de que existen varias maneras de implementar cada una de ellas. La tabla 1.6 presenta algunos ejemplos, con el fin de que sirvan de guía al alumno.

## 1.5. Estructura de un programa en lenguaje ensamblador

Para explicar la estructura de un programa en lenguaje ensamblador y las distintas directivas del ensamblador utilizaremos como guía el sencillo programa descrito en el cuadro 1.

Lo primero que podemos ver en el listado del cuadro 1 es que un programa en lenguaje ensamblador no es más que un texto estructurado en líneas con el siguiente formato:

```
etiqueta: <instrucción o directiva>    @ comentario
```

Cada uno de estos campos es opcional, es decir, podemos tener por ejemplo líneas con instrucción pero sin etiqueta ni comentario.

El fichero que define el programa comienza con una serie de órdenes (directivas de ensamblado) dedicadas a definir dónde se van a almacenar los datos, ya sean datos de entrada o de salida. A continuación aparece el código del programa escrito con instrucciones de del repertorio ARM. Como el ARM es una máquina Von Neuman los datos y las instrucciones

Tabla 1.6: Ejemplos de implementaciones de algunas estructuras de control habituales.

Pseudocódigo	Ensamblador
<pre> <b>if</b> (R1 &gt; R2)     R3 = R4 + R5; <b>else</b>     R3 = R4 - R5 sigue la ejecución normal </pre>	<pre> <b>CMP</b> R1, R2 <b>BLE</b> else <b>ADD</b> R3, R4, R5 <b>B</b> fin_if else: <b>SUB</b> R3, R4, R5 fin_if: sigue la ejecución normal </pre>
<pre> <b>for</b> (i=0; i&lt;8; i++) {     R3 = R1 + R2; } sigue la ejecución normal </pre>	<pre> for: <b>MOV</b> R0, #0          @R0 actúa como índice i       <b>CMP</b> R0, #8       <b>BGE</b> fin_for       <b>ADD</b> R3, R1, R2       <b>ADD</b> R0, R0, #1       <b>B</b> for fin_for: sigue la ejecución normal </pre>
<pre> <b>do</b> {     R3 = R1 + R2;     i = i + 1; } <b>while</b>( i != 8 ) sigue la ejecución normal </pre>	<pre> do: <b>MOV</b> R0, #0          @R0 actúa como índice i      <b>ADD</b> R3, R1, R2      <b>ADD</b> R0, R0, #1      <b>CMP</b> R0, #8      <b>BNE</b> do sigue la ejecución normal </pre>
<pre> <b>while</b> (R1 &lt; R2) {     R2= R2-R3; } sigue la ejecución normal </pre>	<pre> while: <b>CMP</b> R1, R2        <b>BGE</b> fin_w        <b>SUB</b> R2, R2, R3        <b>B</b> while fin_w: sigue la ejecución normal </pre>

utilizan el mismo espacio de memoria. Como programa informático (aunque esté escrito en lenguaje ensamblador), los datos de entrada estarán definidos en unas direcciones de memoria, y los datos de salida se escribirán en direcciones de memoria reservadas para ese fin. De esa manera si se desean cambiar los valores de entrada al programa se tendrán que cambiar a mano los valores de entrada escritos en el fichero. Para comprobar que el programa funciona correctamente se tendrá que comprobar los valores almacenados en las posiciones de

---

**Cuadro 1** Ejemplo de programa en ensamblador de ARM.
 

---

```

.global start

.equ UNO, 0x01

.data
DOS: .word 0x02

.bss
RES: .space 4

.text
start:
    MOV R0, #UNO
    LDR R1, =DOS
    LDR R2, [R1]
    ADD R3, R0, R2
    LDR R4, =RES
    STR R3, [R4]
FIN: B .
.end
  
```

---

memoria reservadas para la salida una vez se haya ejecutado el programa.

Los términos utilizados en la descripción de la línea son:

- **etiqueta:** es una cadena de texto que el ensamblador relacionará con la dirección de memoria correspondiente a ese punto del programa. Si en cualquier otro punto del programa se utiliza esta cadena en un lugar donde debiese ir una dirección, el ensamblador sustituirá la etiqueta por el modo de acceso correcto a la dirección que corresponde a la etiqueta. Por ejemplo, en el programa del cuadro 1 la instrucción **LDR R1,=DOS** carga en el registro R1 el valor de la etiqueta **DOS**, es decir, la dirección en la que hemos almacenado nuestra variable **DOS**, actuando a partir de este momento el registro R1 como si fuera un puntero. Debemos notar aquí que esta instrucción no se corresponde con ningún formato de **ldr** válido. Es una pseudo-instrucción, una facilidad que nos da el ensamblador, para poder cargar en un registro un valor inmediato o el valor de un símbolo o etiqueta. El ensamblador reemplazará esta instrucción por un **ldr** válido que cargará de memoria el valor de la etiqueta **DOS**, aunque necesitará ayuda del enlazador para conseguirlo, como veremos más adelante.
- **instrucción:** el mnemotécnico de una instrucción de la arquitectura destino (algunas de las descritas en la sección 1.4, ver figura 1.5). A veces puede estar modificado por el uso de etiquetas o macros del ensamblador que faciliten la codificación. Un ejemplo es el caso descrito en el punto anterior donde la dirección de un load se indica mediante una etiqueta y es el ensamblador el que codifica esta dirección como un registro base más un desplazamiento.
- **directiva:** es una orden al propio programa ensamblador. Las directivas permiten inicializar posiciones de memoria con un valor determinado, definir símbolos que hagan

más legible el programa, marcar el inicio y el fin del programa, etc (ver figura 1.4). Debemos tener siempre en cuenta que, aparte de escribir el código del algoritmo mediante instrucciones de ensamblador, el programador en lenguaje ensamblador debe reservar espacio en memoria para las variables, y en caso de que deban tener un valor inicial, escribir este valor en la dirección correspondiente. Las directivas más utilizadas son:

- **.global**: exporta un símbolo para que pueda utilizarse desde otros ficheros, resolviéndose las direcciones en la etapa de enlazado. El comienzo del programa se indica mediante la directiva **.global start**, y dicha etiqueta debe aparecer otra vez justo antes de la primera instrucción del programa, para indicar dónde se encuentra la primera instrucción que el procesador debe ejecutar.
- **.equ**: define un símbolo con un valor. De forma sencilla podemos entender un símbolo como una cadena de caracteres que será sustituida allí donde aparezca por un valor, que nosotros definimos. Por ejemplo, **.equ UNO, 0x01** define un símbolo **UNO** con valor **0x01**. Así, cuando en la línea **MOV R0, #UNO** se utiliza el símbolo, el ensamblador lo sustituirá por su valor.
- **.word**: se suele utilizar para inicializar las variables de entrada al programa. Inicializa la posición de memoria actual con el valor indicado tamaño palabra (también podría utilizarse **.byte**). Por ejemplo, en el programa del cuadro 1 la línea **DOS: .word 0x02** inicializa la posición de memoria con el valor **0x02**, donde **0x** indica hexadecimal.
- **.space**: reserva espacio en memoria tamaño byte para guardar las variables de salida, si éstas no se corresponden con las variables de entrada. Siempre es necesario indicar el espacio que se quiere reservar. Por ejemplo, en el programa del cuadro 1 la línea **RES: .space 4** reserva cuatro bytes (una palabra (word)) que quedan sin inicializar. La etiqueta **RES** podrá utilizarse en el resto del programa para referirse a la dirección correspondiente a esta palabra.
- **.end**: Finalmente, el ensamblador dará por concluido el programa cuando encuentre la directiva **.end**. El texto situado detrás de esta directiva de ensamblado será ignorado.
- Secciones. Normalmente el programa se estructura en secciones, generalmente **.text**, **.data** y **.bss**. Para definir estas secciones simplemente tenemos que poner una línea con el nombre de la sección. A partir de ese momento el ensamblador considerará que debe colocar el contenido subsiguiente en la sección con dicho nombre.
  - **.bss**: es la sección en la que se reserva espacio para almacenar el resultado.
  - **.data**: es la sección que se utiliza para declarar las variables con valor inicial
  - **.text**: contiene el código del programa.
- **comentario**: una cadena de texto para comentar el código. Con **@** se comenta hasta el final de la línea actual. Pueden escribirse comentarios de múltiples líneas como comentarios C (entre **/\*** y **\*/**).

Las líneas que contienen directamente una instrucción serán codificadas correctamente como una instrucción de la arquitectura objetivo, ocupando así una palabra de memoria (4 bytes). Cuando se utiliza una *facilidad* del ensamblador, éste puede sustituirla por más de una instrucción, ocupando varias palabras. Si la línea es una directiva que implica reserva

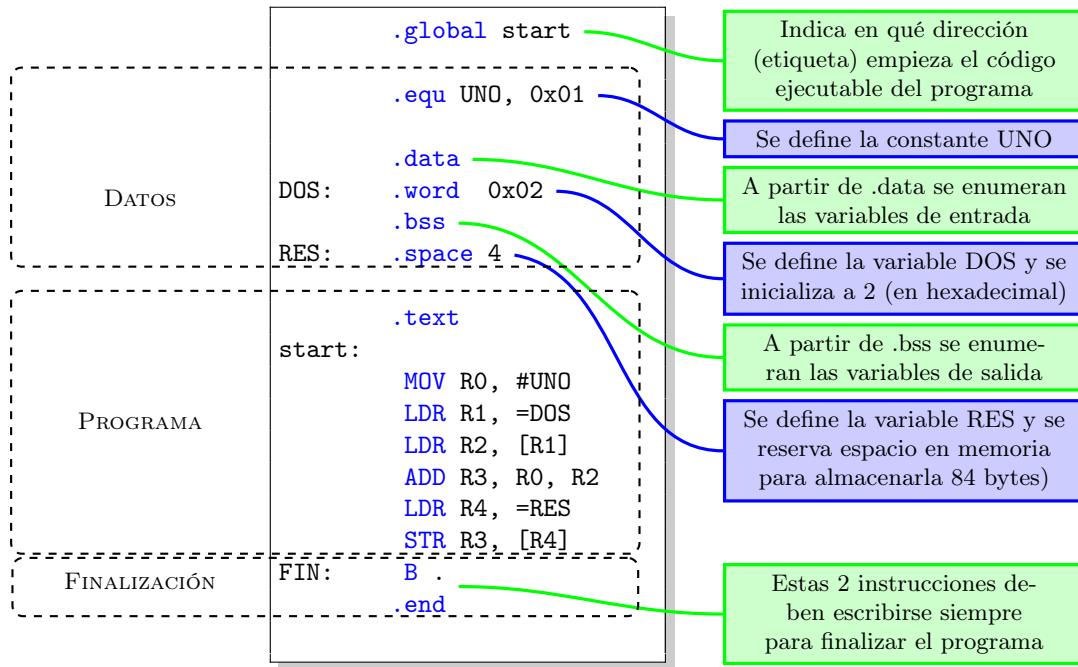


Figura 1.4: Descripción de la estructura de un programa en ensamblador: datos.

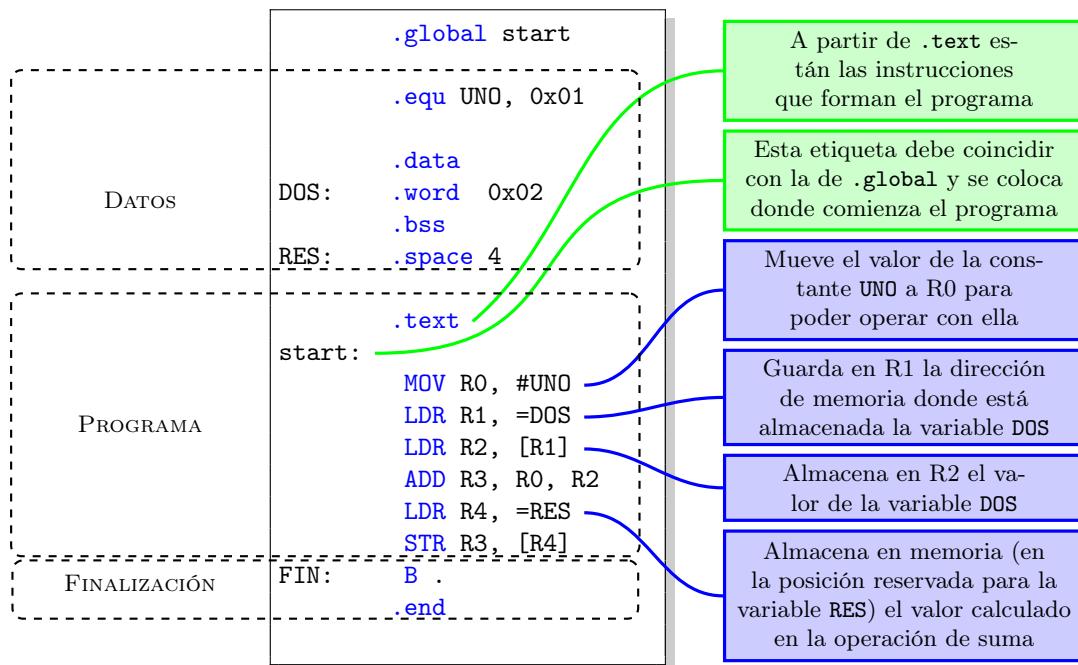


Figura 1.5: Descripción de la estructura de un programa en ensamblador: instrucciones.

de memoria el ensamblador reservará esta memoria y colocará después la traducción de las líneas subsiguientes.

Los pasos seguidos por el entorno de compilación sobre el código presentado en el Cuadro 1 se resumen en la siguiente tabla:

Tabla 1.7: Traducción de código ensamblador a binario

Código Ensamblador	Código objeto desensamblado	Codificación (Hexadecimal)
<b>start:</b>		
MOV R0, #UNO	MOV r0, #1	e3a00001
LDR R1, =DOS	LDR r1, [pc, #16]	e59f1010
LDR R2, [R1]	LDR r2, [r1]	e5912000
ADD R3, R0, R2	ADD r3, r0, r2	e0803002
LDR R4, =RES	LDR r4, [pc, #8]	e59f4008
STR R3, [R4]	STR r3, [r4]	e5843000
<b>FIN:</b>	B .	

En la Tabla 1.7 el código de la primera columna se corresponde con el código en lenguaje ensamblador tal y como lo programamos. Utilizamos etiquetas para facilitarnos una escritura rápida del algoritmo sin tener que realizar cálculos relativos a la posición de las variables en memoria para realizar su carga en el registro asignado a dicha variable. En la columna del centro aparece el código ensamblador generado por el programa ensamblador al procesar el código de la izquierda. Vemos que ya todas las instrucciones tienen la forma correcta del repertorio ARM. Las etiquetas utilizadas en el código anterior se han traducido por un valor relativo al contador de programa (el dato se encuentra X posiciones por encima o por debajo de la instrucción actual). Una vez obtenido este código desambiguado ya se puede realizar una traducción directa a ceros y unos, que es el único lenguaje que entiende el procesador, esa traducción está presente en la columna de la derecha, donde por ejemplo los bits más significativos se corresponden con el código de condición, seguidos del código de operación de cada una de las instrucciones.

## 1.6. Introducción al entorno de desarrollo

En este laboratorio vamos a utilizar Eclipse como Entorno de Desarrollo Integrado (IDE). Es una aplicación con interfaz gráfica de usuario que nos permitirá desarrollar tanto en lenguaje ensamblador como en C, y depurar sobre un simulador de la arquitectura ARM o depurar en circuito sobre una placa con un procesador de ARM.

Eclipse nos ofrece principalmente la interfaz gráfica y la gestión de los proyectos. Para realizar el resto de tareas hace uso de otras herramientas de GNU: el ensamblador (`as`), el compilador (`gcc`), el enlazador (`ld`) y el depurador (`gdb`). Además, debemos tener en cuenta que desarrollamos sobre un PC para un entorno con procesador ARM, y por tanto necesitamos hacer *compilación cruzada*. Para distinguir las herramientas cruzadas de las nativas del PC suele añadírselas un prefijo que describe la arquitectura objetivo para la que se compila, en nuestro caso este prefijo es: `arm-none-eabi-`. Estas herramientas pueden

utilizarse también directamente desde un interprete de línea de comandos (terminal en Linux o Mac OS X o cmd en Windows). En cualquier caso, debemos siempre emplear la sintaxis y reglas de programación propias de estas herramientas.

### 1.6.1. Creación de un proyecto en Eclipse

Para todas nuestras prácticas deberemos crear un proyecto Eclipse para compilación y depuración cruzadas utilizando el plugin *GNU ARM*. Para ello seguiremos los siguientes pasos:

1. Abrimos Eclipse haciendo doble click sobre el icono del escritorio.
2. Al iniciarse la aplicación nos mostrará una ventana de selección de **workspace**, como la que la Figura 1.6. Debemos seleccionar un **workspace** propio, que no es más que un directorio en el que Eclipse guardará información sobre los proyectos de todas nuestras prácticas. Para que el ordenador del laboratorio vaya lo más ágil posible, es conveniente que pongamos nuestro **workspace** en la carpeta C:\hlocal. Es importante también que siempre guardemos una copia de nuestro **workspace** en un lugar seguro, porque el directorio **hlocal** es local al puesto y común para todos los usuarios.
3. Una vez seleccionado se abrirá la ventana principal de Eclipse. Si acabamos de crear el **workspace** entonces el aspecto será como el que se muestra en la Figura 1.7.
4. Debemos cerrar la pestaña **Welcome** haciendo click en la cruz de la pestaña. Entonces la ventana quedará como la que muestra la Figura 1.8, es la perspectiva C/C++ de Eclipse, que está organizada de la siguiente manera:
  - Panel Izquierdo: el explorador de proyectos. Aparecerán todos los proyectos que tengamos en el **workspace**, cuando los tengamos.
  - Panel central: el editor . Nos permitirá editar los ficheros que contendrán el código fuente de nuestros programas.
  - Panel derecho. Nos permite explorar los símbolos del proyecto activo (funciones, variables, etc).
  - Panel inferior. Tiene varias pestañas, entre las que destacan la pestaña de errores de compilación y la consola. Desde la primera podemos ver los errores y saltar a la línea de código fuente que los causó haciendo click sobre el error. En la segunda podemos ver los comandos que ejecuta Eclipse para hacer la compilación.
5. Para crear el proyecto seleccionamos **File→New→C Project**, con lo que se abrirá una ventana como la de la Figura 1.9. Como indica la figura, seleccionamos las opciones **ARM Cross Target Application**, **Empty Project** y **ARM GCC (Sourcery G++ Lite)**. Elegimos el nombre del proyecto y pulsamos **Finish**. Tendremos el proyecto vacío visible en el explorador de proyectos.
6. Ahora vamos a añadirle un fichero con el código fuente de nuestro primer programa en ensamblador. Para ello seleccionamos **File→New→Source File**, con lo que se abrirá una ventana como la de la Figura 1.10. Para que Eclipse tome el fichero como un fichero fuente con código ensamblador le ponemos extensión **.asm** o **.S**. Una vez creado, haciendo doble click sobre el fichero en el panel izquierdo se abrirá una pestaña en editor, en la que copiamos el código del cuadro 2.

7. Antes de configurar la compilación de nuestro proyecto vamos a añadir a él un fichero más, que servirá para indicar al enlazador cómo debe construir el mapa de memoria del ejecutable final. Aprovechamos para ver otra forma de añadir un fichero al proyecto. En el explorador del proyecto seleccionamos el proyecto y pulsamos el botón derecho del ratón, y seleccionamos **New→File**. Se abrirá una ventana como la de la Figura 1.11, seleccionamos el proyecto y ponemos **ld\_script.1d** como nombre del fichero. Una vez creado, lo abrimos en el editor y copiamos el contenido del cuadro 3.
8. Finalmente debemos configurar el proyecto para que la compilación se realice correctamente. Para ello seleccionamos el proyecto en el panel izquierdo, pulsamos el botón derecho del ratón y seleccionamos la entrada **Properties** en la parte inferior del desplegable. Con ello se abrirá una ventana como la que muestra la Figura 1.12. En esta ventana seleccionamos **C/C++ Build→Settings** y
  - Debemos comprobar que en **Target Processor** está seleccionado **arm7tdmi** como procesador, no están marcadas ninguna de las casillas **Thumb\*** y está seleccionada la opción **Little Endian**.
  - Debemos seleccionar **ARM Sourcery GCC C Linker→General**, y en la casilla **Script file (-T)** debemos escribir la ruta al fichero **ld\_script.1d** que hemos añadido al fichero. Podemos seleccionarlo gráficamente pulsando el botón **Browse**.
9. Compilamos el proyecto. Para ello seleccionamos **Project→Build Project**. Acabado este paso habremos obtenido el ejecutable final, con extensión **.elf** (*Executable Linked Format*, que se encontrará en el subdirectorio **Debug**, dentro del directorio de proyecto de nuestro **workspace**.

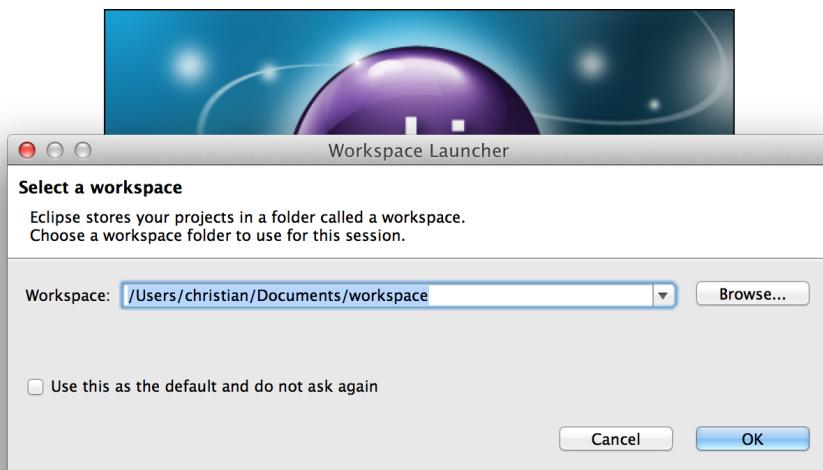


Figura 1.6: Ventana de selección de **workspace**.



Figura 1.7: Ventana de eclipse al abrir un workspace vacío.

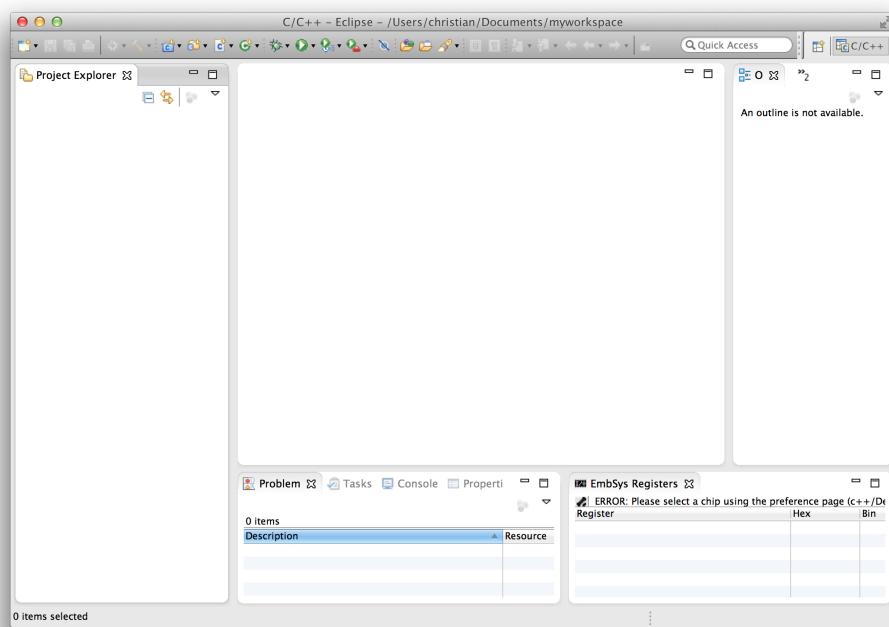


Figura 1.8: Ventana de eclipse con la perspectiva C/C++ sin proyectos.

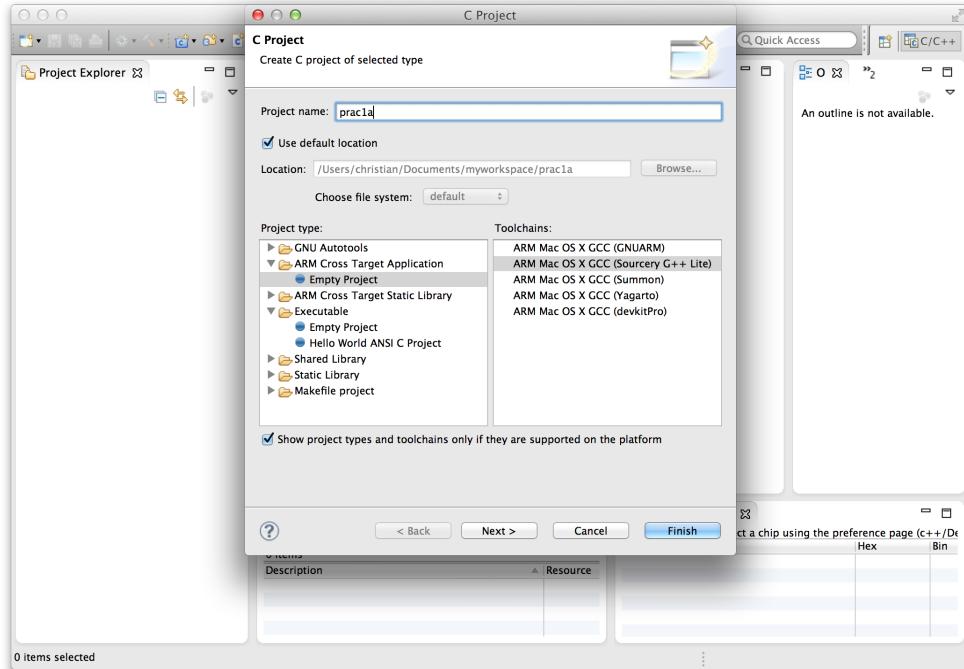


Figura 1.9: Ventana de creación de C project de tipo GNU ARM.

**Cuadro 2** Programa en lenguaje ensamblador que compara dos números y se queda con el mayor.

```
.global start
.data
X:    .word 0x03
Y:    .word 0x0A

.bss
Mayor: .space 4

.text
start:
    LDR R4, =X
    LDR R3, =Y
    LDR R5, =Mayor
    LDR R1, [R4]
    LDR R2, [R3]
    CMP R1, R2
    BLE else
    STR R1, [R5]
    B FIN
else:  STR R2, [R5]
FIN:   B .
.end
```

---

**Cuadro 3** Script de enlazado.

```
SECTIONS
{
    . = 0x0C000000;
    .data : {
        *(.data)
        *(.rodata)
    }
    .bss : {
        *(.bss)
        *(COMMON)
    }
    .text : {
        *(.text)
    }
}
PROVIDE(end = .);
PROVIDE(_stack = 0x0C7FF000 );
}
```

---

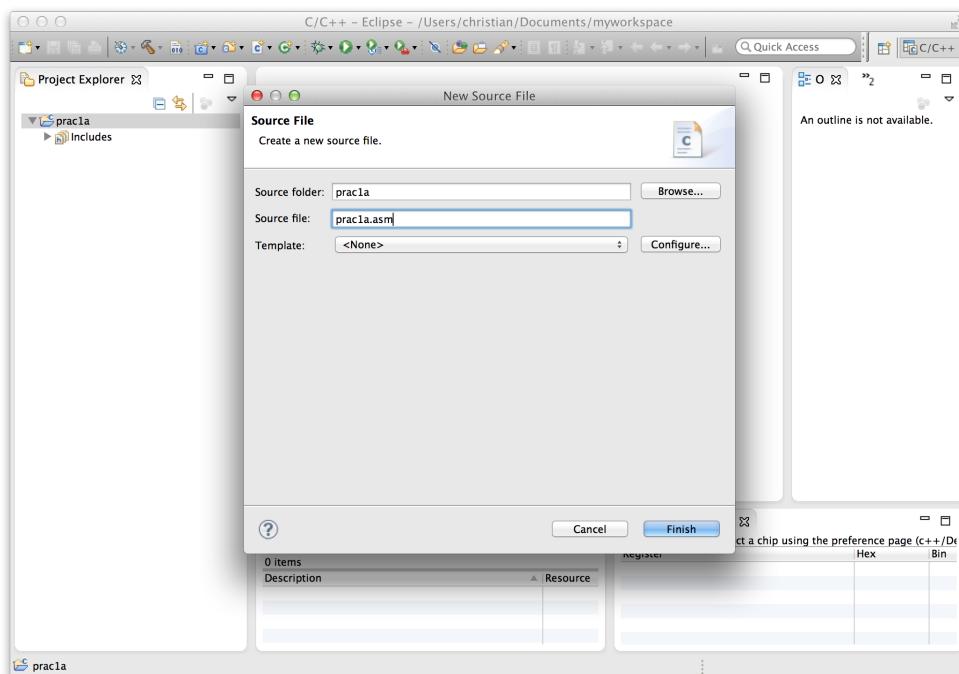


Figura 1.10: Ventana de creación de nuevo fichero fuente.

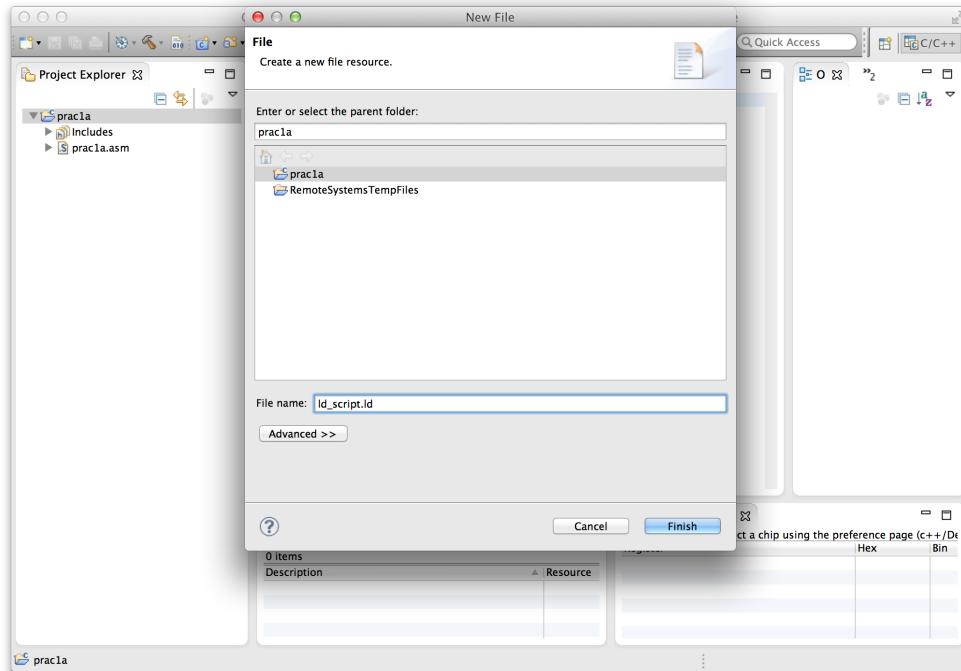


Figura 1.11: Ventana para añadir un nuevo fichero al proyecto.

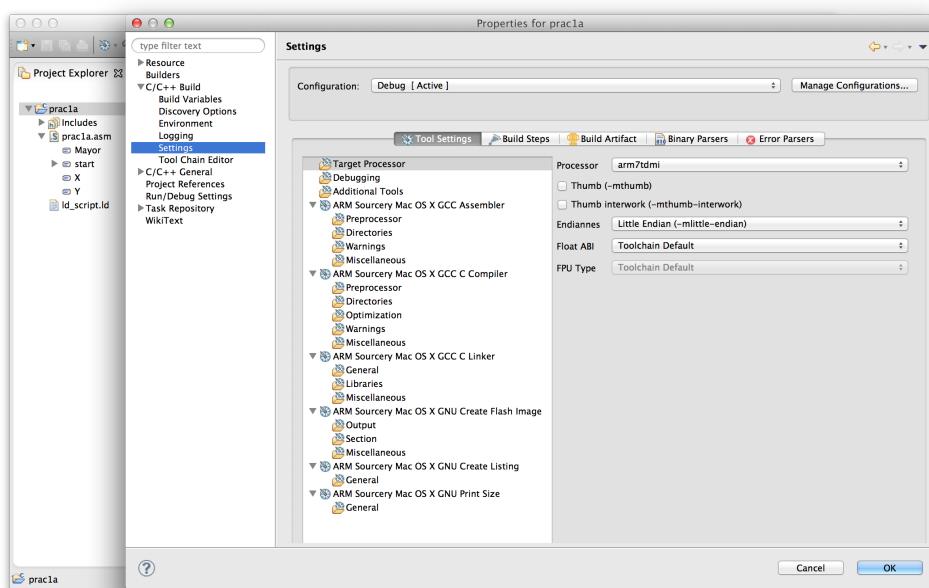


Figura 1.12: Ventana de propiedades (properties) del proyecto.

### 1.6.2. Depuración sobre simulador

Tras generar el ejecutable de nuestro proyecto, nos toca comprobar que funciona correctamente. Para ello usaremos el depurador `arm-none-eabi-gdb` utilizando Eclipse como interfaz. El depurador nos permitirá ejecutar el programa instrucción a instrucción, poner puntos de parada (**breakpoints**), examinar registros, memoria, etc. En estas primeras prácticas no vamos a utilizar un procesador ARM real, configuraremos el depurador para que utilice un simulador.

Hay dos plugins de Eclipse que nos permiten hacer la depuración con GDB sobre el ARM: **GDB Hardware Debugging** y **Zylin Embedded debug**. El primero resulta más sencillo para la depuración en circuito, mientras que el segundo es más sencillo para utilizar con el simulador. Por ello en estas primeras prácticas usaremos **Zylin**.

Para depurar nuestro proyecto seguimos los siguientes pasos:

1. Abrimos la perspectiva **Debug**. Para ello seleccionamos **Window→Open Perspective→Debug**. La apariencia de la ventana de Eclipse cambiará a la mostrada en la Figura 1.13. Como podemos ver, está dividida en varias regiones, cada una de ellas con pestañas:
  - Superior Izquierda: información sobre el proceso de depuración lanzado.
  - Superior Derecha: pestañas donde podemos visualizar los registros, las variables, los breakpoints, ...
  - Central Izquierda: código fuente en depuración. Al principio sólo aparecen los ficheros que tenemos abiertos en la perspectiva C/C++. Se irán abriendo automáticamente nuevas pestañas si el programa en ejecución salta a alguna función definida en otro fichero compilado con símbolos de depuración.
  - Central Derecha: en principio sólo nos muestra una lista de los símbolos definidos por el programa. Es interesante añadir a este panel una pestaña con el código desensamblado. Para ello seleccionamos **Window→Show View→Disassembly**.
  - Inferior: múltiples pestañas con distinto propósito. Por ejemplo aquí podemos poner el visor de memoria, seleccionando **Window→Show View→Memory**.
2. Creamos una configuración de depuración para el proyecto usando el plugin **Zylin**. Para ello seleccionamos **Run→Debug configurations...**, y se abrirá una ventana como la mostrada en la Figura 1.14. En el panel izquierdo seleccionamos **Zylin Embedded debug (Native)** y pulsamos el botón que está en la parte superior izquierda del panel (⊕) para crear la configuración. Deberíamos tener una ventana como la mostrada en la Figura 1.15. Ahora debemos llenar correctamente las siguientes pestañas de la configuración:
  - Debugger: en esta pestaña indicamos el depurador a utilizar. En el laboratorio la ruta al toolchain cruzado se ha añadido al path del usuario, por tanto basta con poner el nombre del depurador en la entrada **GDB debugger**: `arm-none-eabi-gdb`. Además de esto, en la parte superior podemos colocar un breakpoint temporal en donde nosotros queramos. Vamos a colocarlo en la dirección del símbolo `start`, para ello escribimos `*start` en el cuadro **Stop on startup at..**. El resultado se muestra en la Figura 1.16.

- Commands: en esta pestaña damos los comandos que gdb debe ejecutar en la inicialización al iniciar la depuración. En el cuadro **Initialize commands** escribimos **target sim**. Esto indica que el objetivo de depuración es el simulador. En el cuadro **Run commands** escribimos:

```
load
run
```

Esto indica al depurador que cargue el fichero seleccionado en la pestaña principal, leyendo de éste los símbolos de depuración. La ventana resultante se muestra en la Figura 1.17.

- Ya estamos listos para depurar, para ello hacemos click en el botón **Debug**. Esto guardará la configuración de depuración con el nombre seleccionado en la primera pestaña e iniciará una sesión de depuración con esta configuración. Si más adelante queremos volver a depurar este proyecto no será necesario crear una configuración de depuración, bastará con seleccionar la que acabamos de crear. La Figura 1.18 nos muestra el estado inicial que debería tener la ventana en depuración si hemos seguido todos los pasos:

- En la parte izquierda del panel central debemos encontrar el código del cuadro 2, con la primera línea de código tras la etiqueta **start** marcada en verde, con una flecha en el marco izquierdo. Esto quiere decir que el programa ha comenzado correctamente su ejecución simulada y que se ha detenido en el breakpoint que hemos puesto en la dirección de **start**.
- En la parte derecha del panel central debemos encontrar el código desensamblado. Es el contenido de la memoria en el entorno de la dirección de la instrucción actual, reinterpretada como instrucciones.

Notemos la diferencia con el anterior. En el panel izquierdo tenemos el código fuente, tal y como lo programamos, haciendo uso de las facilidades proporcionadas por el ensamblador. En el lado derecho tenemos las instrucciones tal y como las ha generado el ensamblador. Fíjémonos por ejemplo en la instrucción **LDR R4,=X**, que aprovecha una facilidad del ensamblador para escribir en el registro **R4** la dirección correspondiente a la etiqueta **X**. Como podemos ver en el panel derecho, se ha traducido por **LDR R4, [PC, #36]**, que utiliza un modo de direccionamiento correcto para las instrucciones de load en ARM<sup>1</sup>. También podemos observar que la dirección equivalente a **PC+36** aparece como un comentario tras el signo **;**. En cualquiera de los dos paneles podemos poner breakpoints. Si marcamos el botón  en el panel de desensamblado, las instrucciones fuente se mezclarán con las instrucciones máquina correspondientes. Esto facilita el seguimiento del código máquina desensamblado, sobre todo cuando el código fuente es de un lenguaje de alto nivel como C, como veremos en las últimas prácticas.

- Antes de simular nuestro código vamos a abrir una visor de memoria para poder evaluar el valor de nuestras variables. Para ello, lo primero que tenemos que hacer es abrir la pestaña **Memory** en el panel inferior, si no la tenemos abierta ya (**Window→Show View→Memory**). Seleccionamos esta pestaña y añadimos un nuevo monitor pinchando en el ícono . Se abrirá una ventana como la de la Figura 1.19 en la que debemos

---

<sup>1</sup>El proceso de esta traducción implica tanto al ensamblador como al enlazador. El ensamblador pone esta instrucción, que lee un valor de una dirección de memoria donde el enlazador habrá de escribir la dirección correspondiente a la etiqueta **X**. Estudiaremos este proceso en más detalle en la práctica 4.

escribir la dirección a partir de la que queremos monitorizar la memoria. Si ponemos la dirección de descarga, 0x0C000000, en la que hemos colocado la sección `.data`, seguida en orden por las secciones `.bss` y `.text`, el aspecto de la ventana resultante debe ser similar al mostrado en la Figura 1.20.

5. Ahora, para simular todo el código se pude pulsar sobre el icono de **Resume**  o F8 y después pulsar sobre el icono de **Suspend** .

- **¿Cómo sabemos si el código se ha ejecutado correctamente?** El dato mayor se ha escrito en la posición de memoria reservada y etiquetada como **Mayor**. Se puede comprobar su valor en el visor de memoria, que habrá quedado marcado en rojo como ilustra la Figura 1.21.

6. Sin embargo, para entender mejor el funcionamiento de cada una de las instrucciones conviene realizar una ejecución paso a paso. Además, si el resultado no es correcto tendremos que depurar el código para encontrar cuál es la instrucción incorrecta, para lo que una ejecución paso a paso nos será muy útil. Para ello:

- Paramos la simulación pulsando el botón **Terminate**, . Se habilitará el botón **Remove all terminated launches**  que nos permitirá limpiar el panel **Debug** (a veces no se habilita este botón, entonces deberemos pinchar con el botón derecho del ratón en la sesión de depuración y seleccionar **Terminate and Remove**).
- Si necesitamos modificar el código, pasamos a la perspectiva C/C++, editamos los ficheros, los guardamos y recompilamos el proyecto. Luego volvemos a la perspectiva **debug**.
- Podemos iniciar una sesión de depuración con la última configuración de depuración utilizada pulsando el botón .
- Para ejecutar/simular paso a paso tenemos las siguientes opciones:
  - Step Over () : ejecuta hasta la siguiente instrucción. Si es una llamada a subrutina para después de la ejecución completa de la subrutina.
  - Step Into () : ejecuta hasta la siguiente instrucción. Si es una llamada a subrutina se detiene en la primera instrucción de la subrutina.
  - Poner un **Breakpoint** (punto de parada) y reanudar la ejecución pulsando el botón **Resume** . Para poner un **Breakpoint** nos ponemos en el margen izquierdo de alguna instrucción (tanto en el panel de código fuente como en el panel de desensamblado) y hacemos un doble click. Aparecerá la marca . Podemos poner varios **Breakpoints**, la ejecución se detendrá en aquel que se alcance primero.

Además, nos interesará observar los cambios que va realizando nuestro programa, para ello:

- Para ver cómo va cambiando el valor de los registros a medida que ejecutamos las instrucciones usaremos el visor de registros, que se encuentra en el panel superior derecho. Conviene seleccionar para este visor, con el botón , el layout horizontal.
- En el visor de memoria veremos los cambios que nuestro programa realice sobre la memoria.

La Figura 1.22 muestra un ejemplo de una sesión de depuración, donde hemos ido ejecutando paso a paso hasta la instrucción siguiente a `BLE else`. Podemos ver que la condición no se cumple y que se pasa a ejecutar el bloque `else`. Podemos ver el estado de los registros en este momento, y como el PC contiene la dirección `0x0C000030` correspondiente a la siguiente instrucción a ejecutar. También podemos ver cómo el panel de desensamblado nos marca en verde las últimas instrucciones ejecutadas, quedando sin marcar las instrucciones en la rama del `then`.

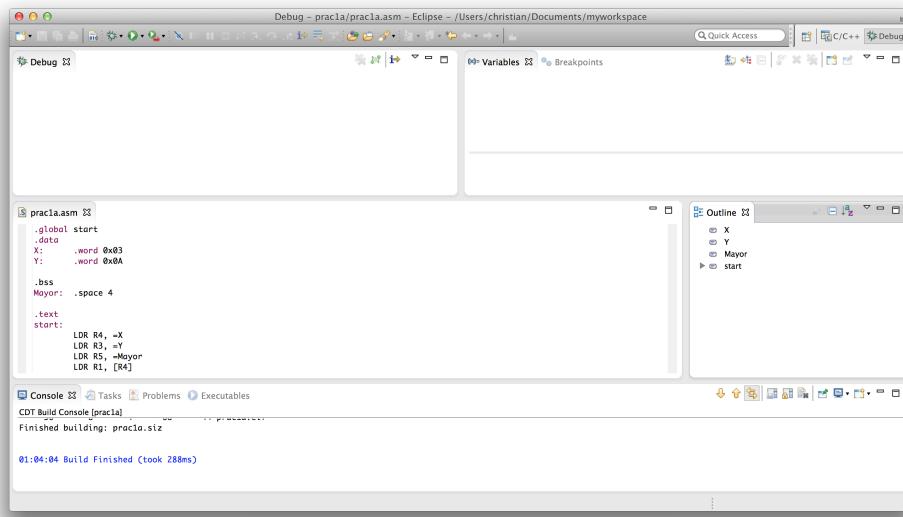


Figura 1.13: Perspectiva de depuración.

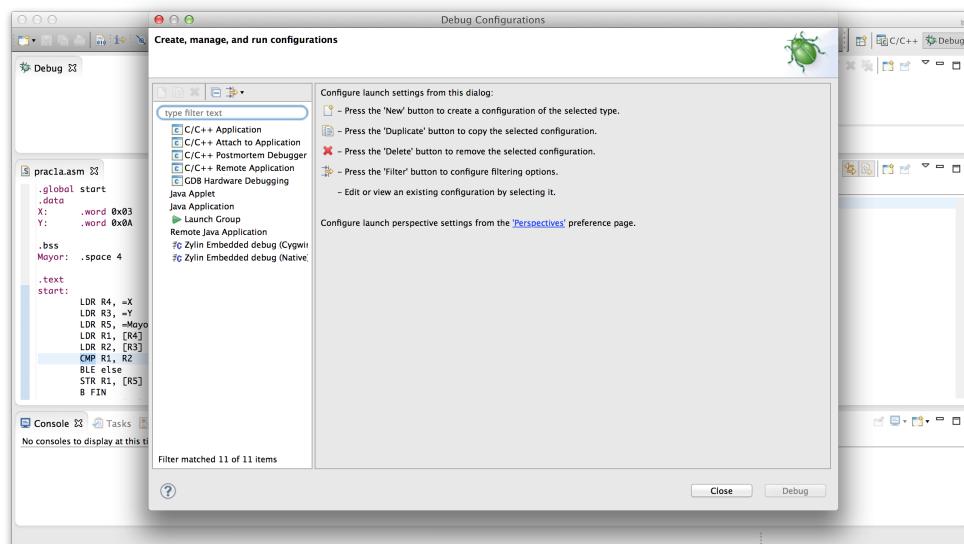


Figura 1.14: Ventana de configuraciones de depuración.

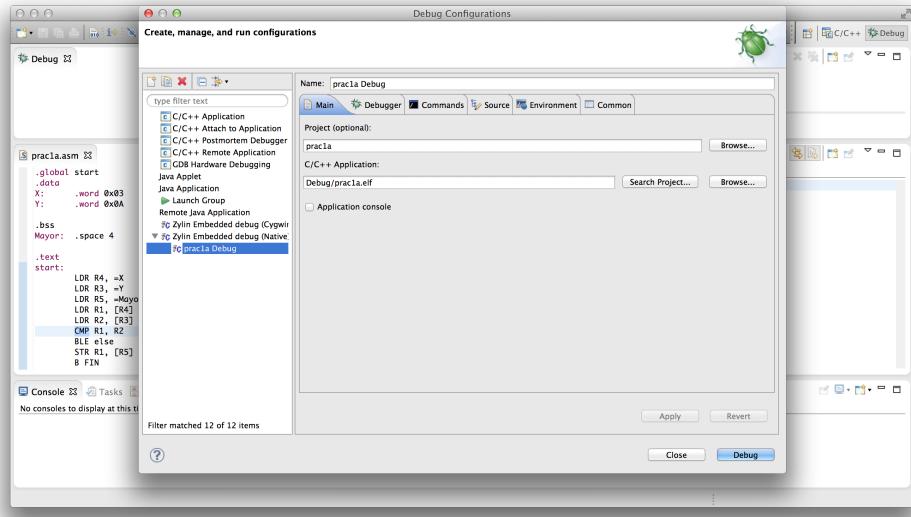


Figura 1.15: Ventana de creación de una nueva configuración Zylin nativa para el proyecto.

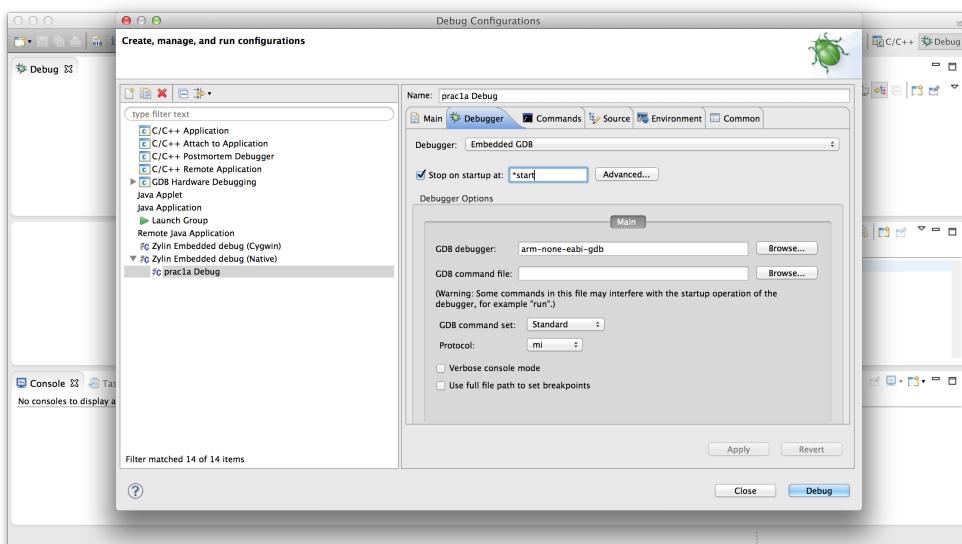


Figura 1.16: Pestaña **debugger** de la configuración de depuración Zylin.

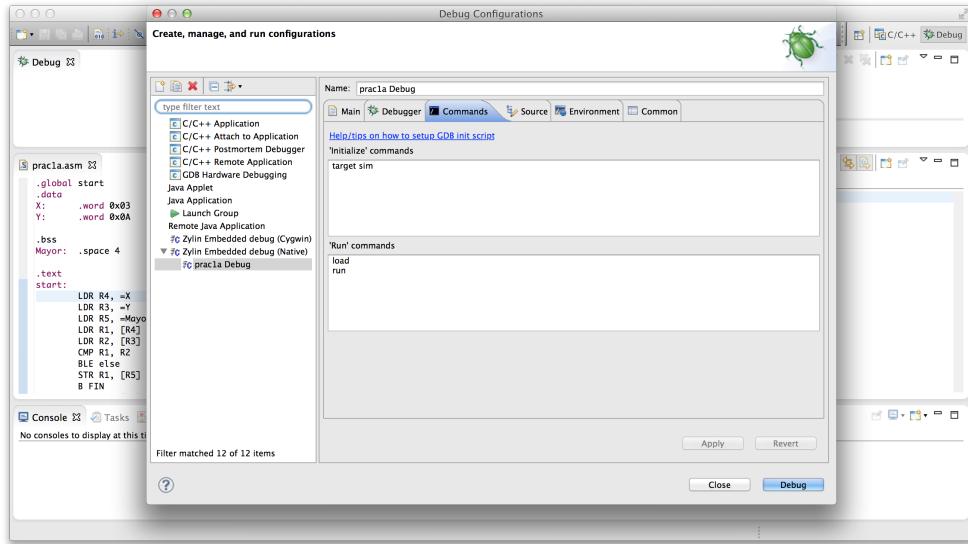


Figura 1.17: Pestaña commands de la configuración de depuración ZYLIN.

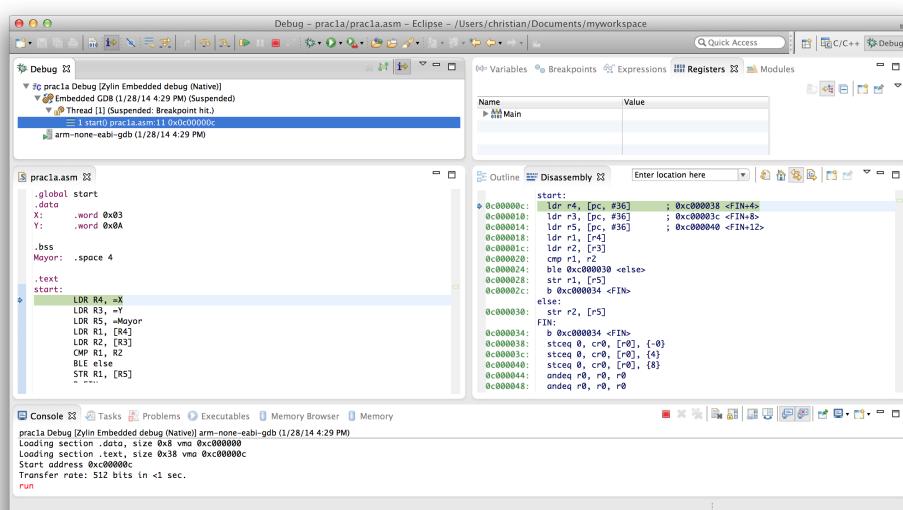


Figura 1.18: Aspecto inicial de la ventana de depuración.

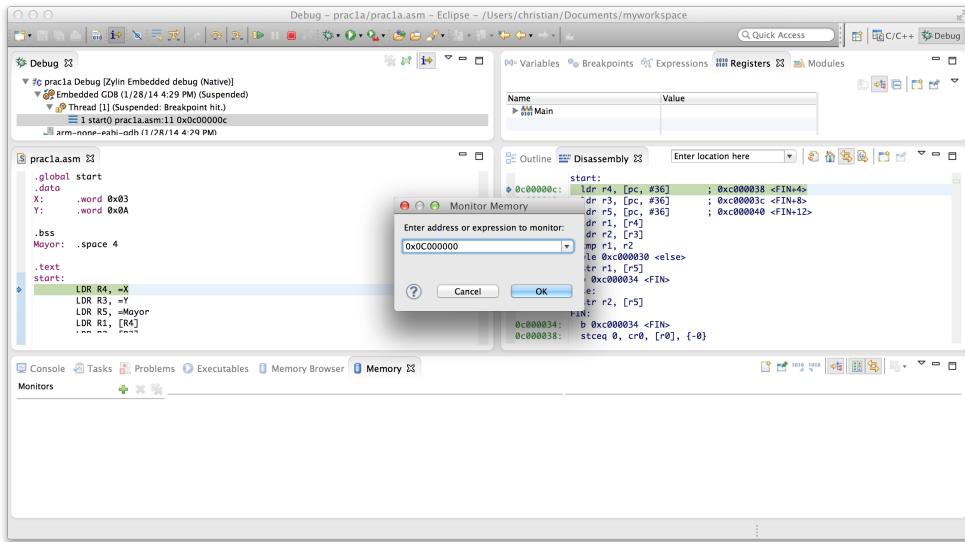


Figura 1.19: Ventana para indicar la dirección del monitor de memoria.

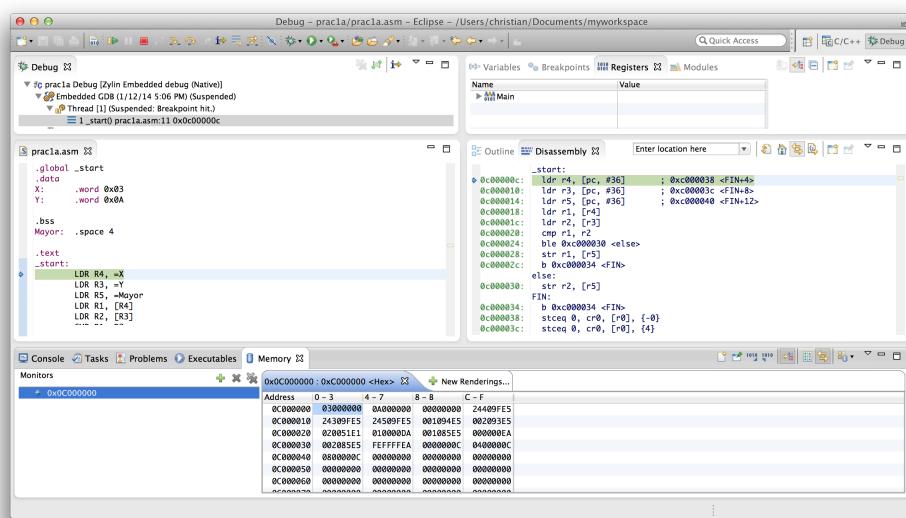


Figura 1.20: Ventana con el aspecto del monitor de memoria en la dirección 0x0C000000.

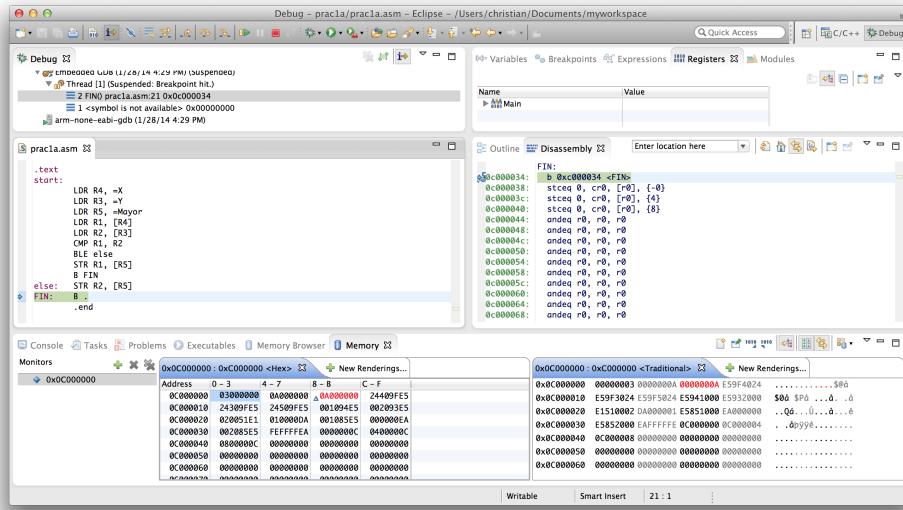


Figura 1.21: Ventana con el aspecto del monitor de memoria tras la simulación completa del programa.

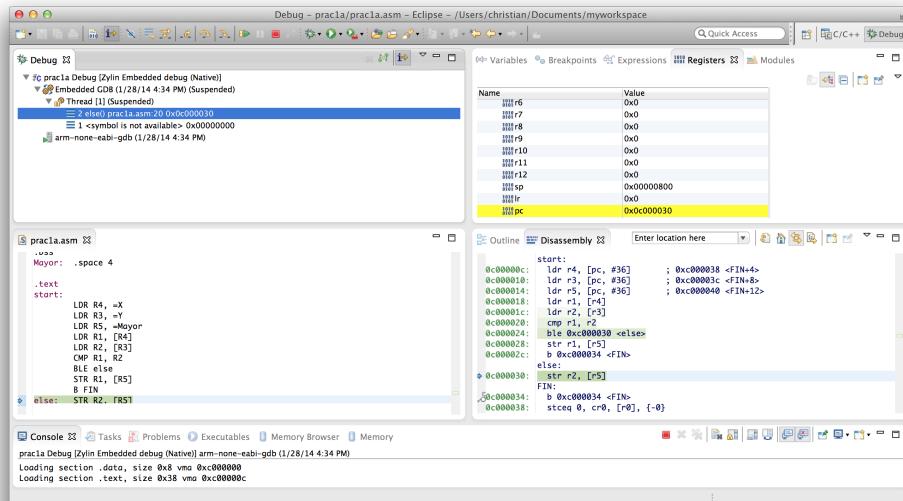


Figura 1.22: Ejemplo de una sesión de depuración con el visor de registros.

## 1.7. Desarrollo de la práctica

El alumno deberá presentar al profesor los siguientes apartados:

1. Desarrollo completo del ejemplo presentado en la sección 1.6.2
2. Desarrollar un programa en código ensamblador del ARM que divida dos números tamaño palabra A y B y escriba el resultado en el número C mediante restas parciales utilizando el algoritmo del cuadro 4.

**Cuadro 4** Pseudocódigo para realizar la división A/B con restas parciales.

---

```
C = 0
while( A >= B ) {
    A = A - B
    C = C + 1
}
```

---